Tests good for SSI and MSI circuits can't cope with the complexity of LSI. New techniques for test generation and response evaluation are required.

LSI Testing Tech

M. S. Abadir*

H. K. Reghbati**

University of Saskatchewan

The growth in the complexity and performance of digital circuits can only be described as explosive. Large-scale integrated circuits are being used today in a variety of applications, many of which require highly reliable operation. This is causing concern among designers of tests for LSI circuits. The testing of these circuits is difficult for several reasons:

- The number of faults that has to be considered is large, since an LSI circuit contains thousands of gates, memory elements, and interconnecting lines, all individually subject to different kinds of faults.

- The observability and controllability of the internal elements of any LSI circuit are limited by the available number of I/O pins. As more and more elements are packed into one chip, the task of creating an adequate test becomes more difficult. A typical LSI chip may contain 5000 gates but only 40 I/O pins.

- The implementation details of the circuits usually are not disclosed by the manufacturer. For example, the only source of information about commercially available microprocessors is the user's manual, which details the instruction set and describes the architecture of the microprocessor at the register-transfer level, with some information on the system timing. The lack of implementation information eliminates the use of many powerful test generation techniques that depend on the actual implementation of the unit under test.

- As more and more gates and flip-flops are packed into one chip, new failure modes—such as pattern-sensitivity faults—arise.[1] These new types of faults are difficult to detect and require lengthy test patterns.

- The dynamic nature of LSI devices requires high-speed test systems that can test the circuits when they are operating at their maximum speeds.

- The bus structure of most LSI systems makes fault isolation more difficult because many devices—any of which can cause a fault—share the same bus.

- Solving the problems above increases the number of test patterns required for a successful test. This in turn increases both the time required for applying that test and the memory needed to store the test patterns and their results.

LSI testing is a challenging task. Techniques that worked well for SSI and MSI circuits, such as the D-algorithm, do not cope with today's complicated LSI and VLSI circuits. New testing techniques must be developed. In what follows, we describe some basic techniques developed to solve the problems associated with LSI testing.

*M. S. Abadir is now in the Department of Electrical Engineering of the University of Southern California.
**H. K. Reghbati is now in the Department of Computing Science of Simon Fraser University, Burnaby, British Columbia.

0272-1732/83/0200-0034$01.00 © 1983 IEEE

niques

Testing methods

There are many test methods for LSI circuits, each with its own way of generating and processing test data. These approaches can be divided into two broad categories—*concurrent* and *explicit*.[2]

In concurrent approaches, normal user-application input patterns serve as diagnostic patterns. Thus testing and normal computation proceed concurrently. In explicit approaches, on the other hand, special input patterns are applied as tests. Hence, normal computation and testing occur at different times.

Concurrent testing. Systems that are tested concurrently are designed such that all the information transferred among various parts of the system is coded with different types of error detecting codes. In addition, special circuits monitor these coded data continuously and signal the detection of any fault.

Different coding techniques are required to suit the different types of information used inside LSI systems. For example, m-out-of-n codes (n-bit patterns with exactly m 1's and $n-m$ 0's) are suitable for coding control signals, while arithmetic codes are best suited for coding ALU operands.[3]

The monitoring circuits—*checkers*—are placed in various locations inside the system so that they can detect most of the faults. A checker is sometimes designed in a way that enables it to detect a fault in its own circuitry as well as in the monitored data. Such a checker is called a *self-checking checker*.[3]

Hayes and McClusky surveyed various concurrent testing methods that can be used with microprocessor-based LSI systems.[2] Concurrent testing approaches provide the following advantages:

- Explicit testing expenses (e.g., for test equipment, down time, and test pattern generation) are eliminated during the life of the system, since the data patterns used in normal operation serve as test patterns.
- The faults are detected instantaneously during the use of the LSI chip, hence the first faulty data pattern caused by a certain fault is detected. Thus, the user can rely on the correctness of his output results within the degree of fault coverage provided by the error detection code used. In explicit approaches, on the other hand, nothing can be said about the correctness of the results until the chip is explicitly tested.
- Transient faults, which may occur during normal operation, are detected if they cause any faulty data pattern. These faults cannot be detected by any explicit testing method.

Unfortunately, the concurrent testing approach suffers from several problems that limit its usage in LSI testing:

- The application patterns may not exercise all the storage elements or all the internal connection lines. Defects may exist in places that are not exercised, and hence the faults these defects would produce will not be detected. Thus, the assumption that faults are detected as

they occur, or at least before any other fault occurs, is no longer valid. Undetected faults will cause fault accumulation. As a result, the fault detection mechanism may fail because most error detection codes have a limited capability for detecting multiple faults.

- Using error detecting codes to code the information signals used in an LSI chip requires additional I/O pins. At least two extra pins are needed as error signal indicators. (A single pin cannot be used, since such a pin stuck at the good value could go undetected.) Because of constraints on pin count, however, such requirements cannot be fulfilled.
- Additional hardware circuitry is required to implement the checkers and to increase the width of the data carriers used for storing and transferring the coded information.
- Designing an LSI circuit for concurrent testing is a much more complicated task than designing a similar LSI circuit that will be tested explicitly.
- Concurrent approaches provide no control over critical voltage or timing parameters. Hence, devices cannot be tested under marginal timing and electrical conditions.
- The degree of fault coverage usually provided by concurrent methods is less than that provided by explicit methods.

The above-mentioned problems have limited the use of concurrent testing for most commercially available LSI circuits. However, as digital systems grow more complex and difficult to test, it becomes increasingly attractive to build test procedures into the UUT—unit under test—itself. We will not consider the concurrent approach further in this article. For a survey of work in concurrent testing, see Hayes and McCluskey.[2]

Explicit testing. All explicit testing methods separate the testing process from normal operation. In general, an explicit testing process involves three steps:

- *Generating the test patterns.* The goal of this step is to produce those input patterns which will exercise the UUT under different modes of operation while trying to detect any existing fault.
- *Applying the test patterns to the UUT.* There are two ways to accomplish this step. The first is external testing—the use of special test equipment to apply the test patterns externally. The second is internal testing—the application of test patterns internally by forcing the UUT to execute a self-testing procedure.[2] Obviously, the second method can only be used with systems that can execute programs (for example, with microprocessor-based systems). External testing gives better control over the test process and enables testing under different timing and electrical conditons. On the other hand, internal testing is easier to use because it does not need special test equipment or engineering skills.

- *Evaluating the responses obtained from the UUT.* This step is designed with one of two goals in mind. The first is the detection of an erroneous response, which indicates the existence of one or more faults (*go/no-go testing*). The other is the isolation of the fault, if one exists, in an easily replaceable module (*fault location testing*). Our interest in this article will be go/no-go testing, since fault location testing of LSI circuits sees only limited use.

Many explicit test methods have evolved in the last decade. They can be distinguished by the techniques used to generate the test patterns and to detect and evaluate the faulty responses (Figure 1). In what follows, we concentrate on explicit testing and present in-depth discussions of the methods of test generation and response evaluation employed with explicit testing.

Test generation techniques

The test generation process represents the most important part of any explicit testing method. Its main goal is to generate those test patterns that, when applied to the UUT, sensitize existing faults and propagate a faulty response to an observable output of the UUT. A test sequence is considered good if it can detect a high percentage of the possible UUT faults; it is considered good, in other words, if its degree of *fault coverage* is high.

Rigorous test generation should consist of three main activities:

- Selecting a good descriptive model, at a suitable level, for the system under consideration. Such a model

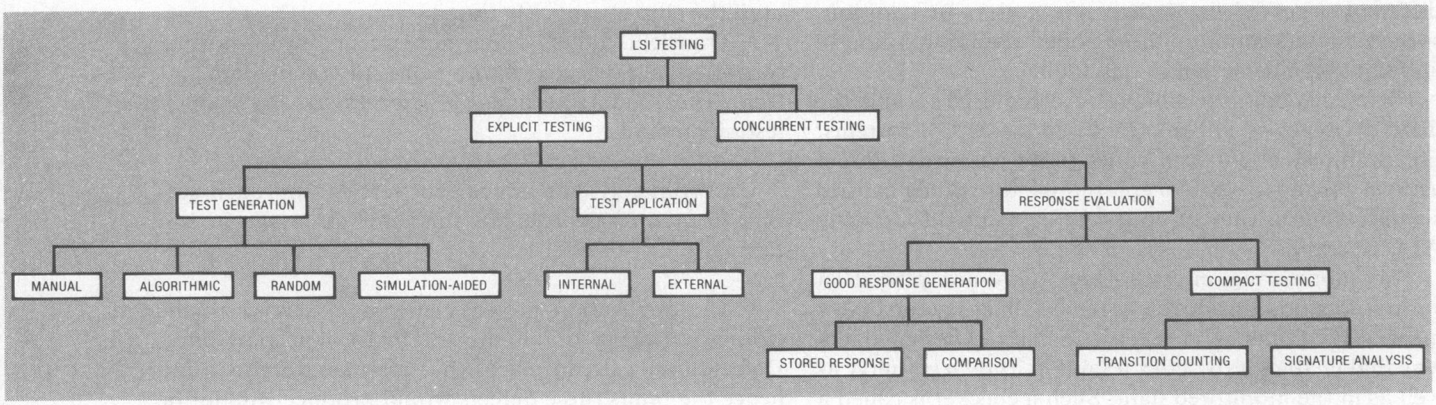


Figure 1. LSI test technology.

NP-complete problems

The theory of NP-completeness is perhaps the most important theoretical development in algorithm research in the past decade.[1] Its results have meaning for all researchers who are developing computer algorithms.

It is an unexplained phenomenon that for many of the problems we know and study, the best algorithms for their solution have computing times which cluster into two groups. The first group consists of problems whose solution is bounded by a polynomial of small degree. Examples include ordered searching, which is $O(\log n)$, polynomial evaluation, which is $O(n)$, and sorting, which is $O(n \log n)$.[2]

The second group contains problems whose best-known algorithms are nonpolynomial. For example, the best algorithms described in Horowitz and Sahni's book[2] for the traveling salesman and the knapsack problems have a complexity of $O(n^2 2^n)$ and $O(2^{n/2})$, respectively. In the quest to develop efficient algorithms, no one has been able to develop a polynomial-time algorithm for any problem in the second group.

The theory of NP-completeness does not provide a method for obtaining polynomial-time algorithms for these problems. But neither does it say that algorithms of this complexity do not exist. What it does show is that many of the problems for which there is no known polynomial-time algorithm are computationally related. In fact, a problem that is NP-complete has the property that it can be solved in polynomial time if all other NP-complete problems can also be solved in polynomial time.

References

1. M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W. H. Freeman, San Francisco, 1978.
2. E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms,* Computer Science Press, Washington, DC, 1978.

should reflect the exact behavior of the system in all its possible modes of operation.

- Developing a fault model to define the types of faults that will be considered during test generation. In selecting a fault model, the percentage of possible faults covered by the model should be maximized, and the test costs associated with the use of the model should be minimized. The latter can be accomplished by keeping the complexity of the test generation low and the length of the tests short. Clearly these objectives contradict one another—a good fault model is usually found as a result of a trade-off between them. The nature of the fault model is usually influenced by the model used to describe the system.
- Generating tests to detect all the faults in the fault model. This part of test generation is the soul of the whole test process. Designing a test sequence to detect a certain fault in a digital circuit usually involves two problems. First, the fault must be *excited;* i.e., a certain test sequence must be applied that will force a faulty value to appear at the fault site if the fault exists. Second, the test must be *made sensitive to* the fault; i.e., the effect of the fault must propagate through the network to an observable output.

Rigorous test generation rests heavily on both accurate descriptive (system) models and accurate fault models.

Test generation for digital circuits is usually approached either at the gate level or at the functional level. The classical approach of modeling digital circuits as a group of connected gates and flip-flops has been used extensively. Using this level of description, test designers introduced many types of fault models, such as the classical stuck-at model. They also assumed that such models could describe physical circuit failures in terms of logic. This assumption has sometimes restricted the number of physical failures that can be modeled, but it has also reduced the complexity of test generation since failures at the elementary level do not have to be considered.

Many algorithms have been developed for generating tests for a given fault in combinational networks.[1,4,5,6,7] However, the complexity of these algorithms depends on the topology of the network; it can become very high for some circuits. Ibarra and Sahni have shown that the problem of generating tests to detect single stuck-at faults in a combinational circuit modeled at the gate level is an NP-complete problem.[8] Moreover, if the circuit is sequential, the problem can become even more difficult depending on the deepness of the circuit's sequential logic.

Thus, for LSI circuits having many thousands of gates, the gate-level approach to the test generation problem is not very feasible. A new approach—the functional level—is needed.

Another important reason for considering faults at the functional level is the constraint imposed on LSI testing by a user environment—the test patterns have to be generated without a knowledge of the implementation details of the chip at the gate level. The only source of information usually available is the typical IC catalog, which details the different modes of operation and describes the general architecture of the circuit. With such information, the test designer finds it easier to define the functional behavior of the circuit and to associate faults with the functions. He can partition the UUT into various modules such as registers, multiplexers, ALUs, ROMs, and RAMs. Each module can be treated as a "black box" performing a specified input/output mapping. These modules can then be tested for *functional failures;* explicit consideration of faults affecting the internal lines is not necessary. The example given below clarifies the idea.

Consider a simple one-out-of-four multiplexer such as the one shown in Figure 2. This multiplexer can be modeled at the gate level as shown in Figure 2a, or at the functional level as shown in Figure 2b.

A possible fault model for the gate-level description is the single stuck-at fault model. With this model, the fault list may contain faults such as the line labeled with "*f*" is stuck at 0, or the control line "C_0" is stuck at 1.

At the functional level, the multiplexer is considered a black box with a well-defined function. Thus, a fault model for it may specify the following as possible faults: selection of wrong source, selection of no source, or presence of stuck-at faults in the input lines or in the multiplexer output. With this model, the fault list may contain faults such as source "X" is selected instead of source "Y," or line "Z" is stuck at 1.

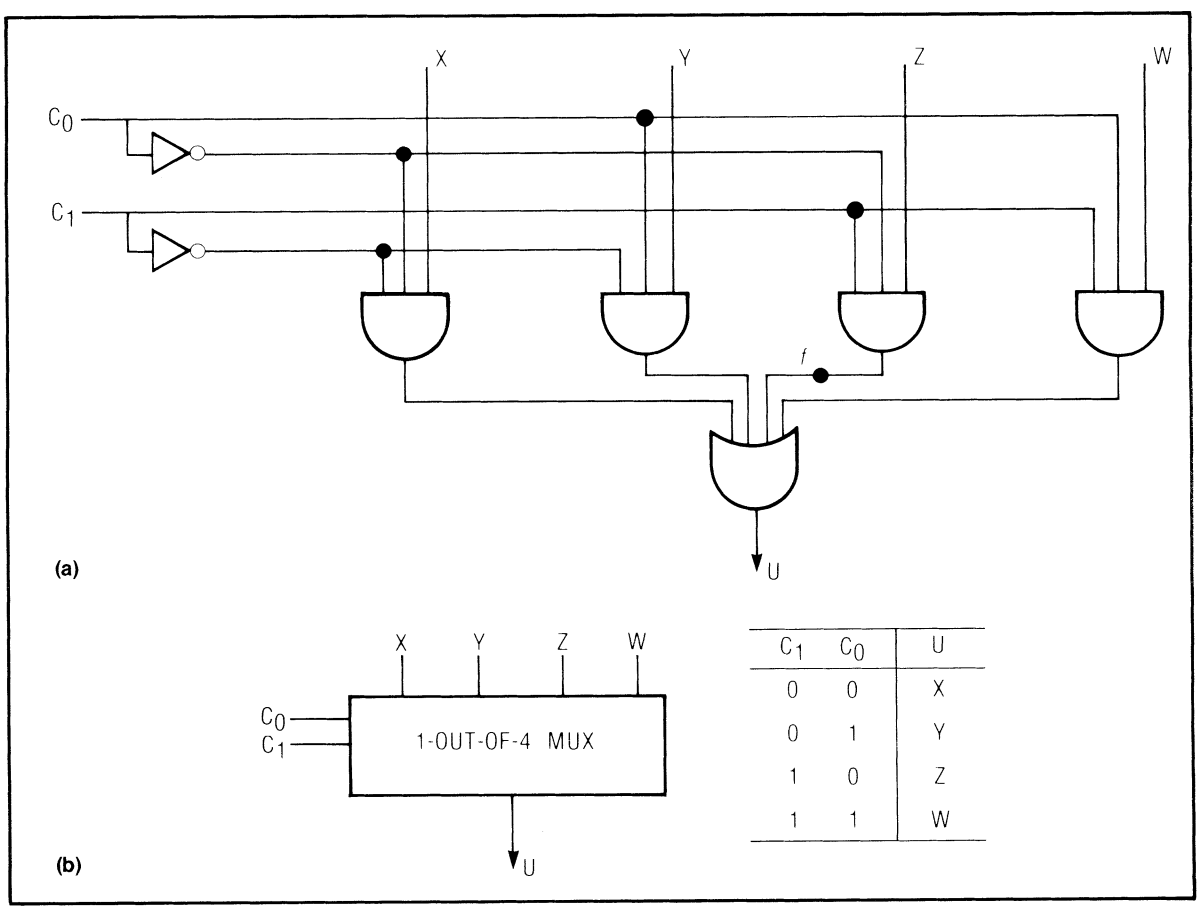
Ad hoc methods—which determine what faults are the most probable—are sometimes used to generate fault lists. But if no fault model is assumed, then the tests derived must be either exhaustive or a rather ad hoc check of the functionality of the system. Exhaustive tests are impossible for even small systems because of the enormous number of possible states, and superficial tests provide neither good coverage nor even an indication of what faults are covered.

Once the fault list has been defined, the next step is to find the test patterns required to detect the faults in the list. As previously mentioned, each fault first has to be excited so that an error signal will be generated somewhere in the UUT. Then this signal has to be sensitized at one of the observable outputs of the UUT. The three examples below describe how to excite and sensitize different types of faults in the types of modules usually encountered in LSI circuits.

Consider the gate-level description of the three-bit incrementer shown in Figure 3. The incrementer output $Y_2Y_1Y_0$ is the binary sum of C_i and the three-bit binary number $X_2X_1X_0$, while C_o is the carry-out bit of the sum. Note that $X_0(Y_0)$ is the least significant bit of the incrementer input (output).

Assume we want to detect the fault "line *f* is stuck at 0." To excite that fault we will force a 1 to appear on line *f* so that, if it is stuck at 0, a faulty value will be generated at the fault site. To accomplish this both X_0 and C_i must be set to 1. To sensitize the faulty 0 at *f*, we have to set X_1 to 1; this will propagate the fault to Y_2 independent of the value of X_2. Note that if we set X_1 to 0, the fault will be masked since the AND gate output will be 0, independent of the value at *f*. Note also that X_2 was not specified in the above test. However, by setting X_2 to 1, the fault will propagate to both Y_2 and C_o, which makes the response evaluation task easier.

Consider a microprocessor RAM and assume we want to generate a test sequence to detect the fault "accessing word *i* in the RAM results in accessing word *j* instead." To excite such a fault, we will use the following sequence



C_1	C_0	U
0	0	X
0	1	Y
1	0	Z
1	1	W

Figure 2. A one-out-of-four multiplexer—gate-level description (a); functional-level description (b).

of instructions (assume a microprocessor with single-operand instructions):

Load the word 00 . . . 0 into the accumulator.
Store the accumulator contents into memory address j.
Load the word 11 . . . 1 into the accumulator.
Store the accumulator contents into memory address i.

If the fault exists, these instructions will force a 11 . . . 1 word to be stored in memory address j instead of 00 . . . 0. To sensitize the fault, we need only read what is in memory address j, using the appropriate instructions. Note that the RAM and its fault have been considered at the functional level, since we did not specify how the RAM is implemented.

Consider the program counter (PC) of a microprocessor and assume we want to generate a test sequence that will detect any fault in the incrementing mode of this PC, i.e., any fault that makes the PC unable to be incremented from x to $x+1$ for any address x. One way to excite this fault is to force the PC to step through all the possible addresses. This can be easily done by initializing the PC to zero and then executing the no-operation instruction $x+1$ times. As a result, the PC will contain an address different than $x+1$. By executing another no-operation instruction, the wrong address can be observed at the address bus and the fault detected. In practice, such an exhaustive test sequence is very expensive, and more economical tests have to be used. Note that, as in the example immediately above, the problem and its solution have been considered at the functional level.

Four methods are currently used to generate test patterns for LSI circuits: manual test generation, algorithmic test generation, simulation-aided test generation, and random test generation.

Manual test generation. In manual test generation, the test designer carefully analyzes the UUT. This analysis can be done at the gate level, at the functional level, or at a combination of the two. The analysis of the different parts of the UUT is intended to determine the specific patterns that will excite and sensitize each fault in the fault list. At one time, the manual approach was widely used for medium- and small-scale digital circuits. Then, the formulation of the D-algorithm and similar algorithms eliminated the need for analyzing each circuit manually and provided an efficient means to generate the required test patterns.[1,5] However, the arrival of LSI circuits and microprocessors required a shift back toward manual test generation techniques, because most of the algorithmic techniques used with SSI and MSI circuits were not suitable for LSI circuits.

Manual test generation tends to optimize the length of the test patterns and provides a relatively high degree of fault coverage. However, generating tests manually takes a considerable amount of effort and requires persons with special skills. Realizing that test generation has to be done economically, test designers are now moving in the direction of automatic test generation.

One good example of manual test generation is the work done by Sridhar and Hayes,[9] who generated test patterns for a simple bit-sliced microprocessor at the functional level.

A bit-sliced microprocessor is an array of n identical ICs called slices, each of which is a simple processor for operands of k-bit length, where k is typically 2 or 4. The interconnections among the n slices are such that the entire array forms a processor for nk-bit operands. The simplicity of the individual slices and the regularity of the interconnections make it feasible to use systematic methods for fault analysis and test generation.

Sridhar and Hayes considered a one-bit processor slice as a simplified model for commercially available bit-sliced processors such as the Am2901.[10] A slice can be modeled as a collection of modules interconnected in a known way. These modules are regarded as black boxes with well-defined input-output relationships. Examples of these functional modules are ALUs, multiplexers, and registers. Combinational modules are described by their truth tables, while sequential modules are defined by their state tables (or state diagrams).

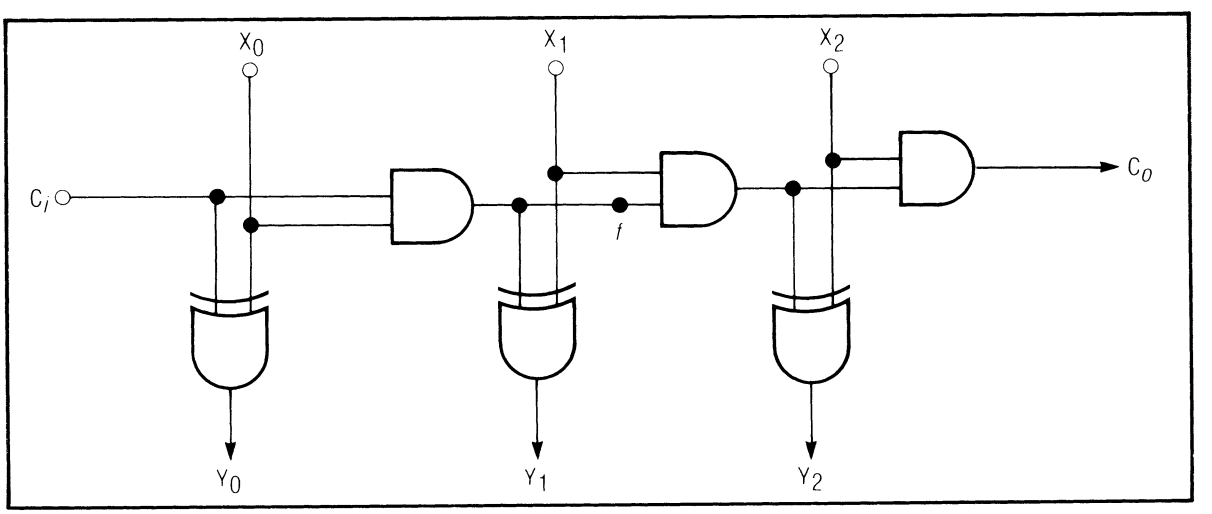


Figure 3. Gate-level description of a three-bit incrementer.

The following fault categories were considered:

- For combinational modules, all possible faults that induce arbitrary changes in the truth table of the module, but that cannot convert it into a sequential circuit.
- For sequential modules, all possible faults that can cause arbitrary changes in the state table of the module without increasing the number of states.

Only one module was assumed to be faulty at any time.

To test for the faults allowed by the above-mentioned fault model, all possible input patterns must be applied to each combinational module (exhaustive testing), and a checking sequence[11] to each sequential module. In addition, the responses of each module must be propagated to observable output lines. The tests required by the individual modules were easily generated manually—a direct consequence of the small operand size ($k = 1$). And because the slices were identical, the tests for one slice were easily extended to the whole array of slices. In fact, Sridhar and Hayes showed that an arbitrary number of simple interconnected slices could be tested with the same number of tests as that required for a single slice, as long as only one slice was faulty at one time. This property is called *C-testability*. Note that the use of carry-lookahead when connecting slices eliminates C-testability. Also note that slices with operand sizes equal to 2 or more usually are not C-testable.

The idea of modeling a digital system as a collection of interconnected functional modules can be used in modeling any LSI circuit. However, using exhaustive tests and checking sequences to test individual modules is feasible only for toy systems. Hence, the fault model proposed by Sridhar and Hayes, though very powerful, is not directly applicable to LSI testing.

Algorithmic test generation. In algorithmic test generation, the test designer devises a set of algorithms to generate the 1's and 0's needed to test the UUT. Algorithmic test techniques are much more economical than manual techniques. They also provide the test designer with a high level of flexibility. Thus, he can improve the fault coverage of the tests by replacing or modifying parts of the algorithms. Of course, this task is much

Path sensitization and the D-algorithm

One of the classical fault detection methods at the gate and flip-flop level is the D-algorithm[1,2] employing the path sensitization testing technique.[3] The basic principle involved in path sensitization is relatively simple. For an input X_i to detect a fault "line a is stuck-at j, $j = 0, 1$," the input X_i must cause the signal a in the normal (fault-free) circuit to take the value $\overline{j}$. This condition is necessary but not sufficient to detect the fault. The error signal must be propagated along some path from its site to an observable output.

To generate a test to detect a stuck-at fault in a combinational circuit, the following path sensitization procedure must be followed:

- Excitation—The inputs must be specified so as to generate the appropriate value (0 for stuck-at 1 and 1 for stuck-at 0) at the site of the fault.
- Error propagation—A path from the fault site to an observable output must be selected, and additional signal values to propagate the fault signal along this path must be specified.
- Line justification—Input values must be specified so as to produce the signals values specified in the step above.

There may be several possible choices for error propagation and line justification. Also, in some cases there may be a choice of ways in which to excite the fault. Some of these choices may lead to an inconsistency, and so the procedure must backtrack and consider the next alternative. If all the alternatives lead to an inconsistency, this implies that the fault cannot be detected.

To facilitate the path sensitization process, we introduce the symbol D to represent a signal which has the value 1 in a normal circuit and 0 in a faulty circuit, and $\overline{D}$ to represent a signal which has the value 0 in a normal circuit and 1 in a faulty circuit. The path sensitization procedure can be formulated in terms of a cubical algebra[1,2] to enable automatic generation of tests. This also facilitates test generation for more complex fault models and for fault propagation through complex logic elements.

We shall define three types of cubes (i.e., line values specified in positional notation):

- For a circuit element E which realizes the combinational function f, the "primitive cubes" offer a typical presentation of the prime implicants of f and $\overline{f}$. These cubes concisely represent the logical behavior of E.
- A "primitive D-cube of a fault" in a logic element E specifies the minimal input conditions that must be applied to E in order to produce an error signal (D or $\overline{D}$) at the output of E.
- The "propagation D-cubes" of a logic element E specify the minimal input conditions to the logic element that are required to propagate an error signal on an input (or inputs) to the output of that element.

To generate a test for a stuck-at fault in a combinational circuit, the D-algorithm must perform the following:

(1) Fault excitation—A primitive D-cube of the fault under consideration must be selected. This generates the error signal D or $\overline{D}$ at the site of the fault. (Usually a choice exists in this step. The initial choice is arbitrary, and it may be necessary to backtrack and consider another choice).

simpler than modifying the 1's and 0's in a manually generated test sequence.

Techniques that use the gate-level description of the UUT, such as path sensitization[4] and the D-algorithm,[5] can no longer be used in testing complicated LSI circuits. Thus, the problem of generating meaningful sets of tests directly from the functional description of the UUT has become increasingly important. Relatively little work has been done on functional-level testing of LSI chips that are not memory elements.[9,12,13,14,15,16,17] Functional testing of memory chips is relatively simple because of the regularity of their design and also because their components can be easily controlled and observed from the outside. Various test generation algorithms have been developed to detect different types of faults in memories.[1,18] In the rest of this section we will concentrate on the general problem of generating tests for irregular LSI chips, i.e., for LSI chips which are not strictly memory chips.

It is highly desirable to find an algorithm that can generate tests for any LSI circuit, or at least most LSI circuits. One good example of work in this area is the technique proposed by Thatte and Abraham for generating tests for microprocessors.[12,13] Another approach, pursued by the authors of this article, is a test generation procedure capable of handling general LSI circuits.[15,16,17]

The Thatte-Abraham technique. Microprocessors constitute a high percentage of today's LSI circuits. Thatte and Abraham[12,13] approached the microprocessor test generation problem at the functional level.

The test generation procedure they developed was based on

- A functional description of the microprocessor at the register-transfer level. The model is defined in terms of data flow among storage units during the execution of an instruction. The functional behavior of a microprocessor is thus described by information about its instruction set and the functions performed by each instruction.
- A fault model describing faults in the various functional parts of the UUT (e.g., the data transfer function, the data storage function, the instruction decoding and control function). This fault model describes the faulty

(2) Implication—In Step 1 some of the gate inputs or outputs may be specified so as to uniquely imply values on other signals in the circuit. The implication procedure is performed both forwards and backwards through the circuit. Implication is performed as follows: Whenever a previously unspecified signal value becomes specified, all the elements associated with this signal are placed on a list B and processed one at a time (and removed). For each element processed, it is determined if new values of 0, 1, D, and $\overline{D}$ are implied, based on the previously specified inputs and outputs. These implied line values are determined by intersecting the test cube (which specifies all the previously determined signal values of the circuit) with the primitive cubes of the element. If any line values are implied, they are specified in the test cube, and the associated gates are placed on the list B. An inconsistency occurs when a value is implied on a line which has been specified previously to a different value. If an inconsistency occurs, the procedure must backtrack to the last point a choice existed, reset all lines to their values at that point, and begin again with the next choice.

(3) D-propagation—All the elements in the circuit whose output values are unspecified and whose input has some signal D or $\overline{D}$ are placed on a list called the D-frontier. In this step, an element from the D-frontier is selected and values are assigned to its unspecified inputs so as to propagate the D or $\overline{D}$ on its inputs to one of its outputs. This is accomplished by intersecting the current test cube describing the circuit signal values with a propagation D-cube of the selected element of the D-frontier, resulting in a new test cube. If such intersection is impossible, a new element in the D-frontier is selected. If intersection fails for all the elements in the D-frontier, the procedure backtracks to the last point at which a choice existed.

(4) Implication of D-propagation—Implication is performed for the new test cube derived in Step 3.

(5) Steps 3 and 4 are repeated until the faulty signal has been propagated to an output of the circuit.

(6) Line justification—Execution of Steps 1 to 5 may result in specifying the output value of an element E but leaving some of the inputs to the element unspecified. The unspecified inputs of such an element are assigned values so as to produce the desired output value. This is done by intersecting the test cube with any primitive cube of the element which has no specified signal values that differ from those of the test cube.

(7) Implication of line justification—Implication is performed on the new test cube derived in Step 6.

(8) Steps 6 and 7 are repeated until all specified element outputs have been justified. Backtracking may again be required.

References

1. J. P. Roth, W. G. Bouricius, and P. R. Schneider, "Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits," *IEEE Trans. Electronic Computers,* Vol. CE-16, No. 5, Oct. 1967, pp. 567-580.
2. M. A. Breuer and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems,* Computer Science Press, Washington, DC, 1976.
3. D. B. Armstrong, "On Finding a Nearly Minimal Set of Fault Detection Tests for Combinatorial Nets," *IEEE Trans. Electronic Computers,* Vol. EC-15, No. 2, Feb. 1966, pp. 63-73.

behavior of the UUT without knowing its implementation details.

The microprocessor is modeled by a graph. Each register in the microprocessor (including general-purpose registers and accumulator, stack, program counter, address buffer, and processor status word registers) is represented by a node of the graph. Instructions of the microprocessor are classified as being of transfer, data manipulation, or branch type. There exists a directed edge (labeled with an instruction) from one node to another if during the execution of the instruction data flow occurs from the register represented by the first node to that represented by the second. Examples of instruction representation are given in Figure 4.

Having described the function or the structure of the UUT, one needs an appropriate fault model in order to derive useful tests. The approach used by Thatte and Abraham is to partition the various functions of a microprocessor into five classes: the register decoding function, the instruction decoding and control function, the data storage function, the data transfer function, and the data manipulation function. Fault models are derived for each of these functions at a higher level and independently of the details of implementation for the microprocessor. The fault model is quite general. Tests are derived allowing any number of faults, but only in one function at a time; this restriction exists solely to cut down the complexity of test generation.

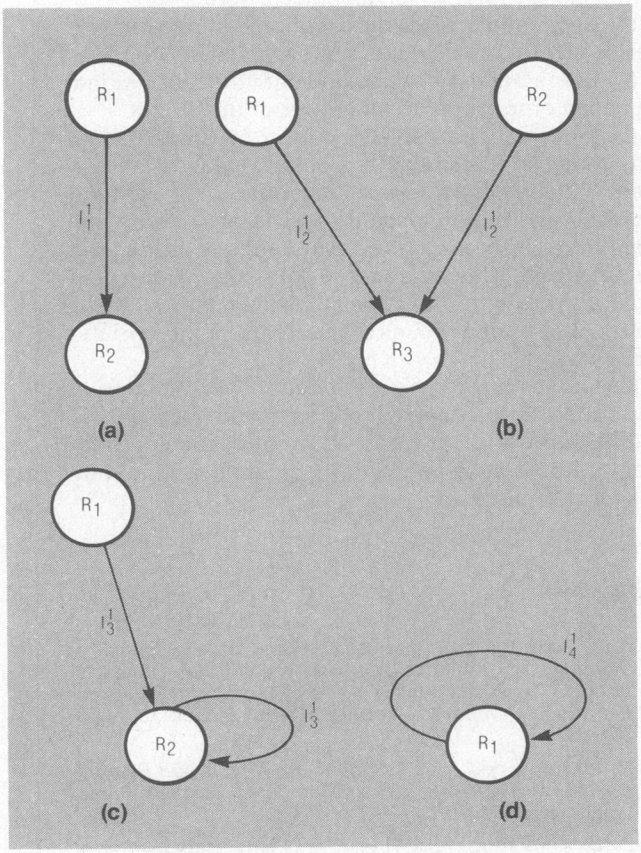


Figure 4. Representations of microprocessor instructions—I_1, transfer instruction, $R_2 \leftarrow R_1$ (a); I_2, add instruction, $R_3 \leftarrow R_1 + R_2$(b); I_3, or instruction, $R_2 \leftarrow R_1$ OR R_2 (c); I_4, rotate left instruction (d).

The fault model for the register decoding function allows any possible set of registers to be accessed instead of a particular register. (If the set is null then no register is accessed.) This fault model is thus very general and independent of the actual realization of the decoding mechanism.

For the instruction decoding and control function, the faulty behavior of the microprocessor is specified as follows—when instruction I_j is executed any one of the following can happen:

- Instead of instruction I_j some other instruction I_k is executed. This fault is denoted by $F(I_j/I_k)$.
- In addition to instruction I_j, some other instruction I_k is activated. This fault is denoted by $F(I_j/I_j+I_k)$.
- No instruction is executed. This fault is denoted by $F(I_j/\Phi)$.

Under this specification, any number of instructions can be faulty.

In the fault model for the data storage function, any cell in any data storage module is allowed to be stuck at 0 or 1. This can occur in any number of cells.

The fault model for the data transfer function includes the following types of faults:

- A line in a path used in the execution of an instruction is stuck at 0 or 1.
- Two lines of a path used in the instruction are coupled; i.e., they fail to carry different logic values.

Note that the second fault type cannot be modeled by single stuck-at faults. The transfer paths in this fault model are logical paths and thus will account for any failure in the actual physical paths.

Since there is a variety of designs for the ALU and other functional units such as increment or shift logic, no specific fault model is used for the data manipulation function. It is assumed that complete test sets can be derived for the functional units for a given fault model.

By carefully analyzing the logical behavior of the microprocessor according to the fault models presented above, Thatte and Abraham formulated a set of algorithms to generate the necessary test patterns. These algorithms step the microprocessor through a precisely defined set of instructions and addresses. Each algorithm was designed for detecting a particular class of faults, and theorems were proved which showed exactly the kind of faults detected by each algorithm. These algorithms employ the excitation and sensitization concepts previously described.

To gain insight into the problems involved in using the algorithms, Thatte investigated the testing of an eight-bit microprocessor from Hewlett-Packard.[12] He generated the test patterns for the microprocessor by hand, using the algorithms. He found that 96 percent of the single stuck-at faults that could affect the microprocessor were detected by the test sequence he generated. This figure indicates the validity of the technique.

The Abadir-Reghbati technique. Here we will briefly describe a test generation technique we developed for LSI circuits.[15,16] We assumed that the tests would be generated in a user environment in which the gate- and flip-flop-level details of the chip were not known.

We developed a module-level model for LSI circuits. This model bypasses the gate and flip-flop levels and directly describes blocks of logic (modules) according to their functions. Any LSI circuit can be modeled as a network of interconnected modules such as counters, registers, ALUs, ROMs, RAMs, multiplexers, and decoders.

Each module in an LSI circuit was modeled as a black box having a number of functions defined by a set of *binary decision diagrams* (see box, next page).[19] This type of diagram, a functional description tool introduced by Akers in 1978, is a concise means for completely defining the logical operation of one or more digital functions in an implementation-free form. The information usually found in an IC catalog is sufficient to derive the set of binary decision diagrams describing the functions performed by the different modules in a device. These diagrams—like truth tables and state tables—are amenable to extensive logical analysis. However, unlike truth tables and state tables, they do not have the unpleasant property of growing exponentially with the number of variables involved. Moreover, the diagrams can be stored and processed easily in a digital computer. An important feature of these diagrams is that they state exactly how the module will behave in every one of its operation modes. Such information can be extracted from the module's diagrams in the form of a set of *experiments*.[15,20] Each of these experiments describes the behavior of the module in one of its modes of operation. The structure of these experiments makes them suitable for use in automatic test generation.

We also developed a functional-level fault model describing faulty behavior in the different modules of an LSI chip. This model is quite independent of the details of implementation and covers functional faults that alter the behavior of a module during one of its modes of operation. It also covers stuck-at faults affecting any input or output pin or any interconnection line in the chip.

Using the above-mentioned models, we proposed a functional test generation procedure based on path sensitization and the D-algorithm.[15] The procedure takes the module-level model of the LSI chip and the functional description of its modules as parameters and generates tests to detect faults in the fault model. The *fault collapsing* technique[1] was used to reduce the length of the test sequence. As in the D-algorithm, the procedure employs three basic operations, namely implication, D-propagation, and line justification. However, these operations are performed on functional modules.

We also presented algorithmic solutions to the problems of performing these operations on functional modules.[16] For each of the three operations, we gave an algorithm which takes the module's set of experiments and current state (i.e., the values assigned to the module inputs, outputs, and internal memory elements) as parameters and generates all the possible states of the module after performing the required operation.

We have also reported our efforts to develop test sequences based on our test generation procedure for typical LSI circuits.[17] More specifically, we considered a one-bit microprocessor slice C that has all the basic features of the four-bit Am2901 microprocessor slice.[10] The circuit C was modeled as a network of eight functional modules: an ALU, a latch register, an addressable register, and five multiplexers. The functions of the individual modules were described in terms of binary decision diagrams or equivalent sets of experiments. Tests capable of detecting various faults covered by the fault model were then generated for the circuit C. We showed that if the fault collapsing technique is used, a significant reduction in the length of the final test sequence results.

The test generation effort was quite straightforward, indicating that the technique can be automated without much difficulty. Our study also shows that for a simplified version of the circuit C the length of the test sequence generated by our technique is very close to the length of the test sequence manually generated by Sridhar and Hayes[9] for the same circuit. We also described techniques for modeling some of the features of the Am2909 four-bit microprogram sequencer[10] that are not covered by the circuit C.

The results of our case study were quite promising and showed that our technique is a viable and effective one for generating tests for LSI circuits.

Simulation-aided test generation. Logic simulation techniques have been used widely in the evaluation and verification of new digital circuits. However, an important application of logic simulation is to interpret the behavior of a circuit under a certain fault or faults. This is known as *fault simulation.* To clarify how this technique can be used to generate tests for LSI systems, we will first describe its use with SSI/MSI-type circuits.

To generate a fault simulator for an SSI/MSI circuit, the following information is needed:[1]

- the gate-level description of the circuit, written in a special language;
- the initial conditions of the memory elements; and
- a list of the faults to be simulated, including classical types of faults such as stuck-at faults and adjacent pin shorts.

The above is fed to a simulation package which generates the fault simulator of the circuit under test. The resulting simulator can simulate the behavior of the circuit under normal conditions as well as when any faults exist.

Now, by applying various input patterns (either generated by hand, by an algorithm, or at random), the simulator checks to see if the output response of the correct circuit differs from one of the responses of the faulty circuits. If it does, then this input pattern detects the fault which created the wrong output response; otherwise the input pattern is useless. If an input pattern is found to detect a certain fault, this fault is deleted from the fault

list and the process continues until either the input patterns or the faults are finished. At the end, the faults remaining in the fault list are those which cannot be detected by the input patterns. This directly measures the degree of fault coverage of the input patterns used.

Two examples of this type of logic simulator are LAMP—the Logic Analyzer for Maintenance Planning developed at Bell Laboratories,[21] and the Testaid III fault simulator developed at the Hewlett-Packard Company.[12] Both work primarily at the gate level and simulate stuck-at faults only. One of the main applications of such fault simulators is to determine the degree of fault coverage provided by a test sequence generated by any other test generation technique.

There are two key requirements that affect the success of any fault simulator:

- the existence of a software model for each primitive element of the circuit, and
- the existence of a good fault model for the UUT which can be used to generate a fault list covering most of the actual physical faults.

These two requirements have been met for SSI/MSI circuits, but they pose serious problems for LSI circuits. If it can be done at all, modeling LSI circuits at the gate level requires great effort. One part of the problem is the lack of detailed information about the internal structure of most LSI chips. The other is the time and memory required to simulate an LSI circuit containing thousands of gates. Another severe problem facing almost all LSI test generation techniques is the lack of good fault models at a level higher than the gate level.

The Abadir-Reghbati description model proposed in the previous section permits the test designer to bypass the gate-level description and, using binary decision diagrams, to define blocks of logic according to their functions. Thus, the simulation of complex LSI circuits can take place at a higher level, and this eliminates the large time and memory requirements. Furthermore, the Abadir-Reghbati fault model is quite efficient and is suitable for simulation purposes. In fact, the implication operation[16] employed by the test generation procedure represents the main building block of any fault simulator. It must be noted that fault simulation techniques are very useful in optimizing the length of the test sequence generated by any test generation technique.

Random test generation. This method can be considered the simplest method for testing a device. A random number generator is used to simultaneously apply random input patterns both to the UUT and to a copy of it known to be fault-free. (This copy is called the *golden unit*.) The results obtained from the two units are compared, and if they do not match, a fault in the UUT is detected. This response evaluation technique is known as comparison testing; we will discuss it later. It is important to note that every time the UUT is tested, a new random test sequence is used.

The important question is how effective the random test is, or, in other words, what fault coverage a random

Binary decision diagrams

Binary decision diagrams are a means of defining the logical operation of digital functions.[1] They tell the user how to determine the output value of a digital function by examining the values of the inputs. Each node in these diagrams is associated with a binary variable, and there are two branches coming out from each node. The right branch is the "1" branch, while the left branch is the "0" branch. Depending on the value of the node variable, one of the two branches will be selected when the diagram is processed.

To see how binary decision diagrams can be used, consider the half-adder shown in Figure 1a. Assume we are interested in defining a procedure to determine the value of C, given the binary values of X and Y. We can do this by looking at the value of X. If X = 0, then

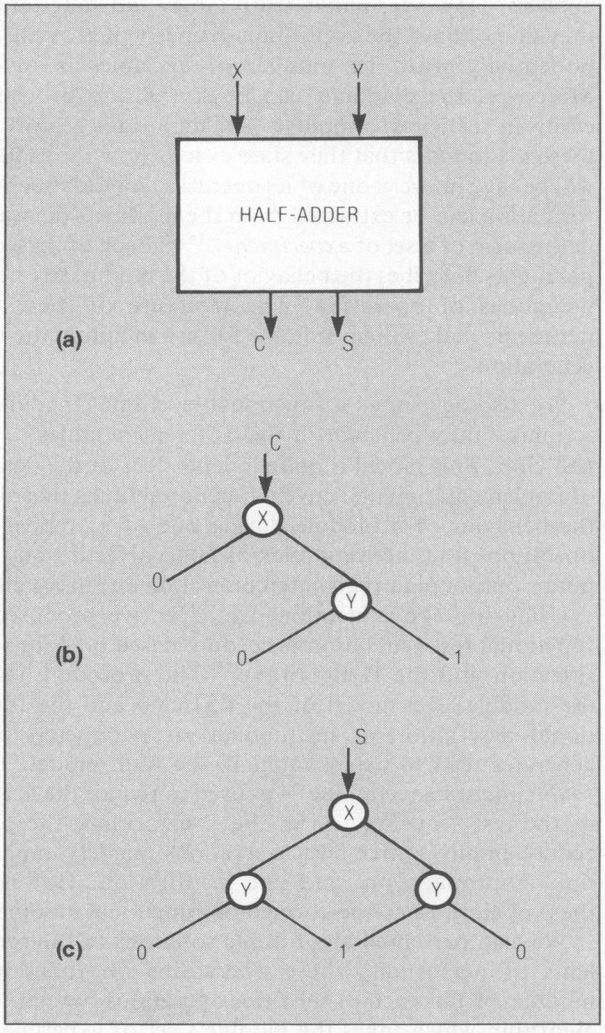


Figure 1. A half-adder (a); binary decision diagram for C = X·Y (b); binary decision diagram for S = X⊕Y (c).

$C = 0$, and we are finished. If $X = 1$, we look at Y. If $Y = 0$, then $C = 0$, else $C = 1$, and in either case we are finished. Figure 1b shows a simple diagram of this procedure. By entering the diagram at the node indicated by the arrow labeled with C and then proceeding through the diagram following the appropriate branches until a 0 or 1 value is reached, we can determine the value of C. Figure 1c shows the diagram representing the function S of the half-adder.

To simplify the diagrams, any diagram node which has two branches as exit branches can be replaced by the variable itself or its complement. These variables are called exit variables. Figure 2 shows how this convention is used to simplify the diagrams describing the half-adder.

In the previous discussion, we have considered only simple diagrams in which the variables within the nodes are primary input variables. However, we can expand the scope of these diagrams by using auxiliary variables as the node variables. These auxiliary variables are defined by their diagrams. Thus, when a user encounters such a node variable, say *g*, while tracing a path, he must first process the diagram defining *g* to determine the value of *g*, and then return to the original node and take the appropriate branch. This process is similar to the use of subroutines in high-level programming languages.

For example, consider the full-adder defined by

$$C_{i+1} = E_i C_i + \overline{E}_i A_i$$

$$S_i = E_i \oplus C_i,$$

where $E_i = A_i \oplus B_i$. Figure 3 shows the diagrams for these three equations. If the user wants to know the value of C_{i+1} when the values of the three primary inputs A_i, B_i, and C are all 1's, he enters the C_{i+1} diagram, where he encounters the node variable E_i. By traversing the E_i diagram, he obtains a value of 0. Returning to the original C_{i+1} diagram with $E_i = 0$ will result in taking the 0 branch and exiting with $C_{i+1} = A_i = 1$.

Since node variables can refer to other auxiliary functions, we can simply describe complex modules by breaking their functions into small subfunctions. Thus, the system diagram will consist of small diagrams connected in a hierarchical structure. Each of these diagrams describes either a module output or an auxiliary variable.

Akers[1] described two procedures to generate the binary decision diagram of a combinational function *f*. The first one uses the truth table description of *f*, while the other uses the boolean expression of *f*. A similar procedure can be derived to generate the binary decision diagram for any sequential function defined by a state table.

Binary decision diagrams can be easily stored and processed by a computer through the use of binary tree structures. Each node can be completely defined by an ordered triple: the node variable and two pointers to the two nodes to which its 0 and 1 branches are directed. Binary decision diagrams can be used in functional testing.[2]

References

1. S. B. Akers, "Binary Decision Diagram," *IEEE Trans. Computers,* Vol. C-27, No. 6, June 1978, pp. 509-516.
2. S. B. Akers, "Functional Testing with Binary Decision Diagrams," *Proc. 8th Int'l Symp. Fault-Tolerant Computing,* June 1978, pp. 82-92.

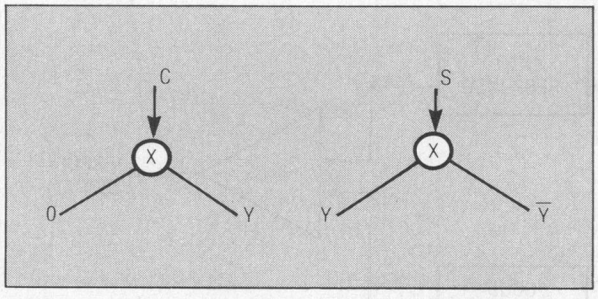


Figure 2. Simplified binary decision diagrams for the half-adder.

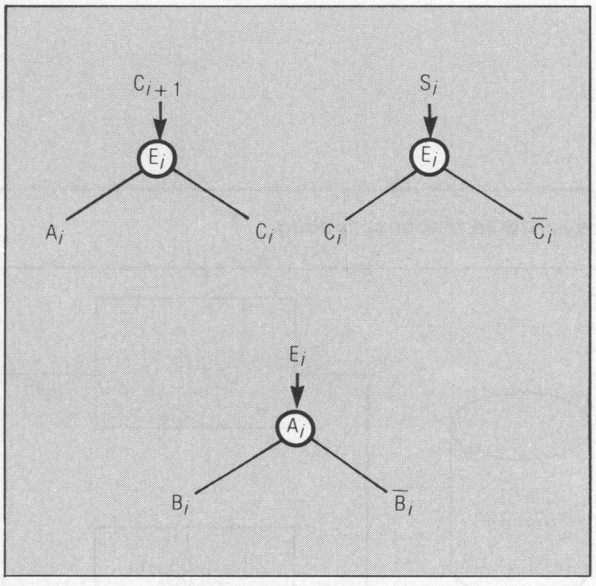


Figure 3. Binary decision diagrams for a full-adder.

test of given length provides. This question can be answered by employing a fault simulator to simulate the effect of random test patterns of various lengths. The results of such experiments on SSI and MSI circuits show that random test generation is most suitable for circuits without deep sequential logic.[1,22,23] However, by combining random patterns with manually generated ones, test designers can obtain very good results.

The increased sequentiality of LSI circuits reduces the applicability of random testing. Again, combining manually generated test patterns with random ones improves the degree of fault coverage. However, two factors restrict the use of the random test generation technique:

- The dependency on the golden unit, which is assumed to be fault-free, weakens the level of confidence in the results.
- There is no accurate measure of how effective the test is, since all the data gathered about random tests are statistical data. Thus, the amount of fault coverage provided by a particular random test process is unpredictable.

Response evaluation techniques

Different methods have been used to evaluate UUT responses to test patterns. We restrict our discussion to the case where the final goal is only to detect faults or, equivalently, to detect any wrong output response. There are two ways of achieving this goal—using a good response generator or using a compact testing technique.

Good response generation. This technique implements an ideal strategy—comparing UUT responses with good response patterns to detect any faulty response. Clearly, the key problems are how to obtain a good response and at what stage in the testing process that response will be generated. In current test systems, two approaches to solving these problems are taken—*stored response testing* and *comparison testing*.

Stored response testing. In stored response testing, a one-shot operation generates the good response patterns at the end of the test generation stage. These patterns are stored in an auxiliary memory (usually a ROM). A flow diagram of the stored response testing technique is shown in Figure 5.

Different methods can be used to obtain good responses of a circuit to a particular test sequence. One way is to do it manually by analyzing the UUT and the test patterns. This method is the most suitable if the test patterns were generated manually in the first place.

The method most widely used to obtain good responses from the UUT is to apply the test patterns either to a known good copy of the UUT—the golden unit—or

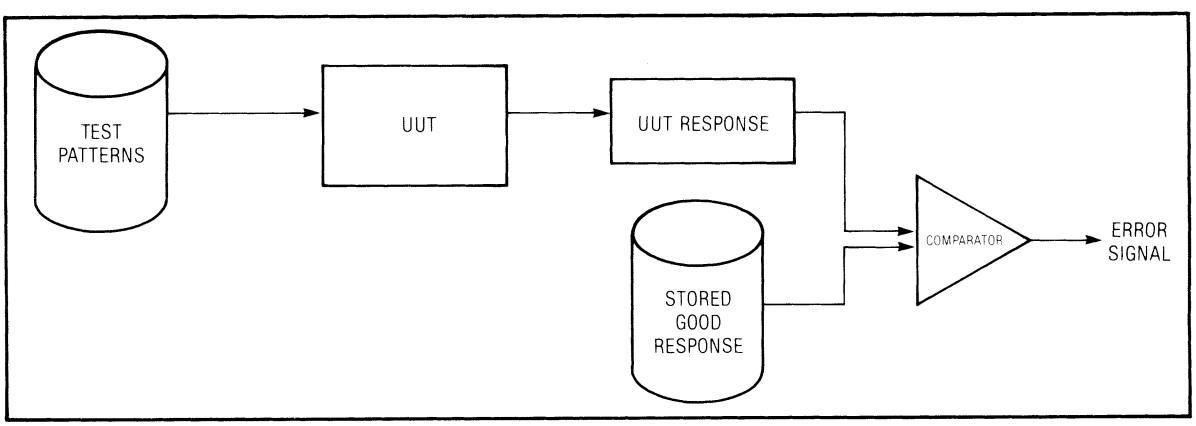


Figure 5. Stored response testing.

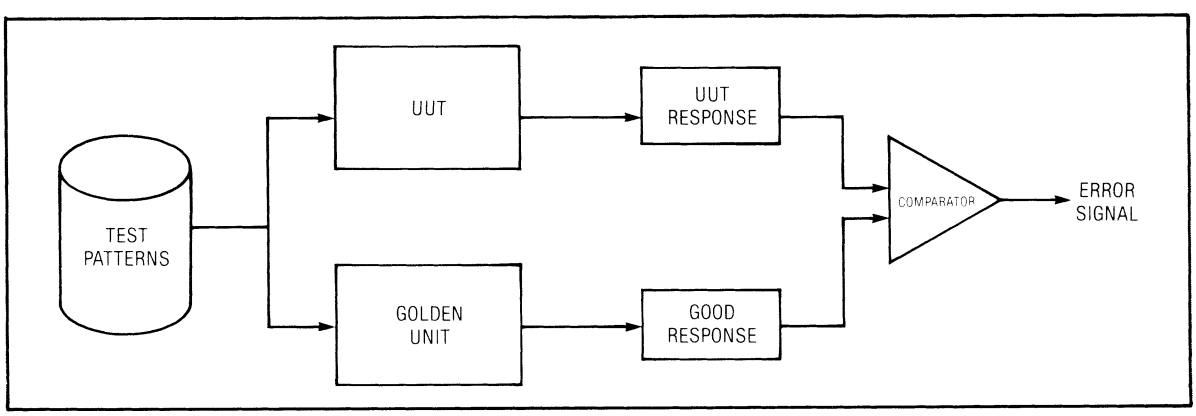


Figure 6. Comparison testing.

to a software-simulated version of the UUT. Of course, if fault simulation techniques were used to generate the test patterns, the UUT's good responses can be obtained very easily as a partial product from the simulator.

The use of a known good device depends on the availability of such a device. Hence, different techniques must be used for the user who wants to test his LSI system and for the designer who wants to test his prototype design. However, golden units are usually available once the device goes into production. Moreover, confidence in the correctness of the responses can be increased by using three or five good devices together to generate the good responses.

The major advantage of the stored response technique is that the good responses are generated only once for each test sequence, thus reducing the cost of the response evaluation step. However, the stored response technique suffers from various disadvantages:

- Any change in the test sequence requires the whole process to be repeated.
- A very large memory is usually needed to store all the good responses to a reasonable test sequence, because both the length and the width of the responses are relatively large. As a result, the cost of the testing equipment increases.
- The speed with which the test patterns can be applied to the UUT is limited by the access time of the memory used to store the good responses.

Comparison testing. Another way to evaluate the responses of the UUT during the testing process is to apply the test patterns simultaneously to both the UUT and a golden unit and to compare their responses to detect any faulty response. The flow diagram of the comparison testing technique is shown in Figure 6. The use of comparison testing makes possible the testing of the UUT at different speeds under different electrical parameters, given that these parameters are within the operating limits of the golden unit, which is assumed to be ideal.

Note that in comparison testing the golden unit is used to generate the good responses every time the UUT is tested. In stored response testing, on the other hand, the golden unit is used to generate the good responses only once.

The disadvantages of depending on a golden unit are more serious here, however, since every explicit testing process requires one golden unit. This means that every tester must contain a golden copy of each LSI circuit tested by that tester.

One of the major advantages of comparison testing is that nothing has to be changed in the response evaluation stage if the test sequence is altered. This makes comparison testing highly desirable if test patterns are generated randomly.

Compact testing. The major drawback of good response generation techniques in general, and stored response testing in particular, is the huge amount of response data that must be analyzed and stored. Compact testing methods attempt to solve this by compressing the response data R into a more compact form $f(R)$ from which most of the fault information in R can be derived. Thus, because only the compact form of the good responses has to be stored, the need for large memory or expensive golden units is eliminated. An important property of the compression function f is that it can be implemented with simple circuitry. Thus, compact testing does not require much test equipment and is especially suited for field maintenance work. A general diagram of the compact testing technique is shown in Figure 7.

Several choices for the function f exist, such as "the number of 1's in the sequence," "the number of 0 to 1 and 1 to 0 transitions in the sequence" (*transition counting*),[24] or "the signature of the sequence" (*signature analysis*).[25] For each compression function f, there is a slight probability that a response R1 different from the fault-free response R0 will be compressed to a form equal to $f(R0)$, i.e., $f(R1) = f(R0)$. Thus, the fault causing the UUT to produce R1 instead of R0 will not be detected, even though it is covered by the test patterns.

The two compression functions that are the most widely accepted commercially are transition counting and signature analysis.

Transition counting. In transition counting, the number of logical transitions (0 to 1 and vice versa) is computed at each output pin by simply running each output of the UUT into a special counter. Thus, the number of counters needed is equal to the number of

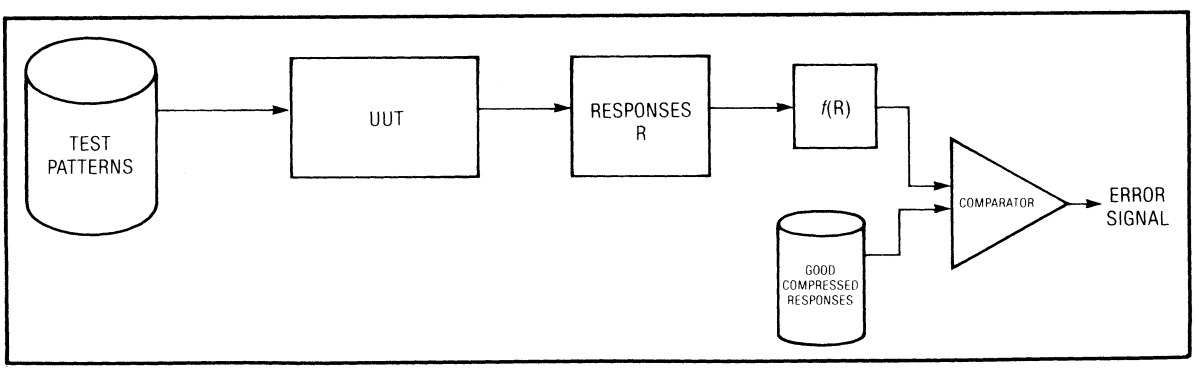
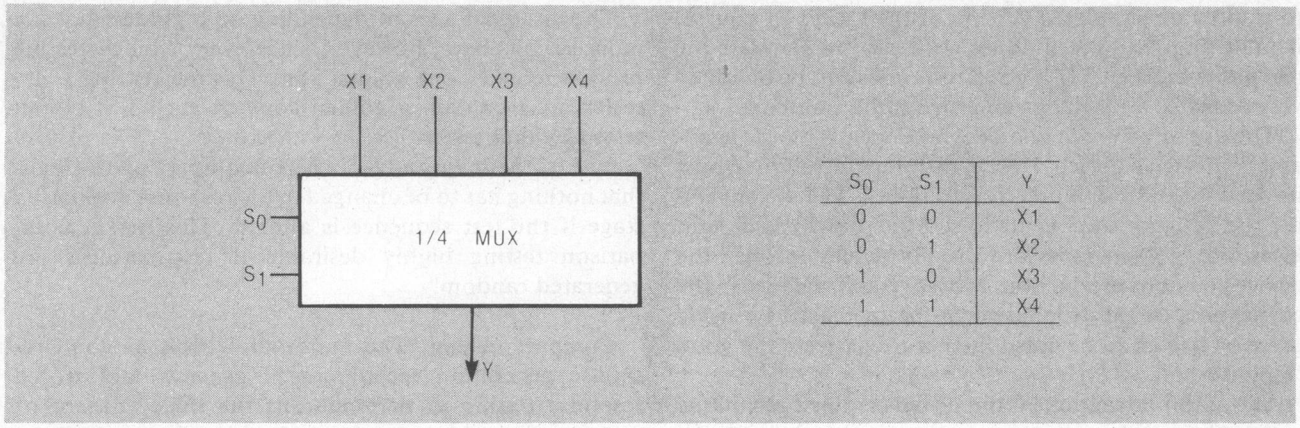


Figure 7. Compact testing.



S_0	S_1	Y
0	0	X1
0	1	X2
1	0	X3
1	1	X4

Figure 8. A one-out-of-four multiplexer.

output pins observed. For every m-bit output data stream (at one pin), an n-bit counter is required, where $n = \lceil \log_2 m \rceil$. As in stored response testing, the transition counts of the good responses are obtained by applying the test sequence to a golden copy of the UUT and counting the number of transitions at each output pin. This latter information is used as a reference in any explicit testing process.

In the testing of an LSI circuit by means of transition counting, the input patterns can be applied to the UUT at a very high rate, since the response evaluation circuitry is very fast. Also, the size of the memory needed to store the transition counts of the good responses can be very small. For example, a transition counting test using 16 million patterns at a rate of one MHz will take 16 seconds, and the compressed stored response will occupy only K 24-bit words, where K is the number of output pins. This can be contrasted with the 16 million K-bit words of storage space needed if regular stored response testing is used.

The test patterns used in a transition counting test system must be designed such that their output responses maximize the fault coverage of the test.[24] The example below shows how this can be done.

Consider the one-out-of-four multiplexer shown in Figure 8. To check for multiple stuck-at faults in the multiplexer input lines, eight test patterns are required, as shown in Table 1. The sequence of applying these eight patterns to the multiplexer is not important if we want to evaluate the output responses one by one. However, this sequence will greatly affect the degree of fault coverage if transition counting is used. To illustrate this fact, consider the eight single stuck-at faults in the four input lines X1, X2, X3, and X4 (i.e., X1 stuck-at 0, X1 stuck-at 1, X2 stuck-at 0, and so on). Each of these faults will be detected by only one pattern among the eight test patterns. For example, the fault "X1 stuck-at 0" will be detected by applying the first test pattern in Table 1, but the other seven test patterns will not detect this fault. Now, suppose we want to use transition counting to evaluate the output responses of the multiplexer. Applying the eight test patterns in the sequence shown in Table 1 (from top to bottom) will produce the output response 10101010 (from left to right), with a transition count of seven. Any possible combination of the eight faults described above will change the transition count to a number different from seven, and the fault will be detected. (Note that no more than four of the eight faults can occur at any one time.) Thus, the test sequence shown in Table 1 will detect all single and multiple stuck-at faults in the four input lines of the multiplexer.

Now, if we change the sequence of the test patterns to the one shown in Table 2, the fault coverage of the test will decrease considerably. The output responses of the sequence of Table 2 will be 11001100, with a transition count of three. As a result, six of the eight single stuck-at

Table 1.
The eight test patterns used for testing the multiplexer of Figure 8.

S_0	S_1	X1	X2	X3	X4	Y
0	0	1	0	0	0	1
0	0	0	1	1	1	0
0	1	0	1	0	0	1
0	1	1	0	1	1	0
1	0	0	0	1	0	1
1	0	1	1	0	1	0
1	1	0	0	0	1	1
1	1	1	1	1	0	0

Table 2.
A different sequence of the eight multiplexer test patterns.

S_0	S_1	X1	X2	X3	X4	Y
0	0	1	0	0	0	1
0	1	0	1	0	0	1
0	0	0	1	1	1	0
0	1	1	0	1	1	0
1	0	0	0	1	0	1
1	1	0	0	0	1	1
1	0	1	1	0	1	0
1	1	1	1	1	0	0

faults will not be detected, because the transition count of the six faulty responses will remain three. For example, the fault "X1 stuck-at 1" will change the output response to 11101100, which has a transition count of three. Hence, this fault will not be detected. Moreover, most of the multiple combinations of the eight faults will not change the transition count of the output, and hence they will not be detected either.

It is clear from the above example that the order of applying the test patterns to the UUT greatly affects the fault coverage of the test. When testing combinational circuits, the test designer is completely free to choose the order of the test patterns. However, he cannot do the same with test patterns for sequential circuits. More seriously, because he is dealing with LSI circuits that probably have multiple output lines, he will find that a particular test sequence may give good results at some outputs and bad results at others. One way to solve these contradictions is to use simulation techniques to find the optimal test sequence. However, because of the limitations discussed here, transition counting cannot be recognized as a powerful compact LSI testing method.

Signature analysis. In 1977 Hewlett-Packard Corporation introduced a new compact testing technique called signature analysis, intended for testing LSI systems.[25-28] In this method, each output response is passed through a 16-bit linear feedback shift register whose contents $f(R)$, after all the test patterns have been applied, are called the test *signature*. Figure 9 shows an example of a linear feedback shift register used in signature analysis.

The signature provided by linear feedback shift registers can be regarded as a unique fingerprint—hence, test designers have extremely high confidence in these shift registers as tools for catching errors. To better understand this confidence, let us examine the 16-bit linear feedback shift register shown in Figure 9. Let us assume a data stream of length n is fed to the serial data input line (representing the output response to be evaluated). There are 2^n possible combinations of data streams, and each one will be compressed to one of the 2^{16} possible signatures. Linear feedback shift registers have the property of equally distributing the different combinations of data streams over the different signatures.[27] This property is illustrated by the following numerical examples:

- Assume $n = 16$. Then each data stream will be mapped to a distinctive signature (one-to-one mapping).
- Assume $n = 17$. Then exactly two data streams will be mapped to the same signature. Thus, for a particular data stream (the UUT good output response), there is only one other data stream (a faulty output response) that will have the same signature; i.e., only one faulty response out of $2^{17} - 1$ possible faulty responses will not be detected.
- Assume $n = 18$. Then four different data streams will be mapped to the same signature. Hence, only three faults out of $2^{18} - 1$ possible faults will not be detected.

We can generalize the results obtained above. For any response data stream of length $n > 16$, the probability of missing a faulty response when using a 16-bit signature analyzer is[27]

$$\frac{2^{n-16}-1}{2^n-1} \cong 2^{-16}, \text{ for } n >> 16.$$

Hence, the possibility of missing an error in the bit stream is very small (on the order of 0.002 percent). Note also that a great percentage of the faults will affect more than one output pin—hence the probability of not detecting these kind of faults is even lower.

Signature analysis provides a much higher level of confidence for detecting faulty output responses than that provided by transition counting. But, like transition counting, it requires only very simple hardware circuitry and a small amount of memory for storing the good signatures. As a result, the signatures of the output responses can be calculated even when the UUT is tested at its maximum speed. Unlike transition counting, the degree of fault coverage provided by signature analysis is not sensitive to the order of the test patterns. Thus, it is clear that signature analysis is the most attractive solution to the response evaluation problem.

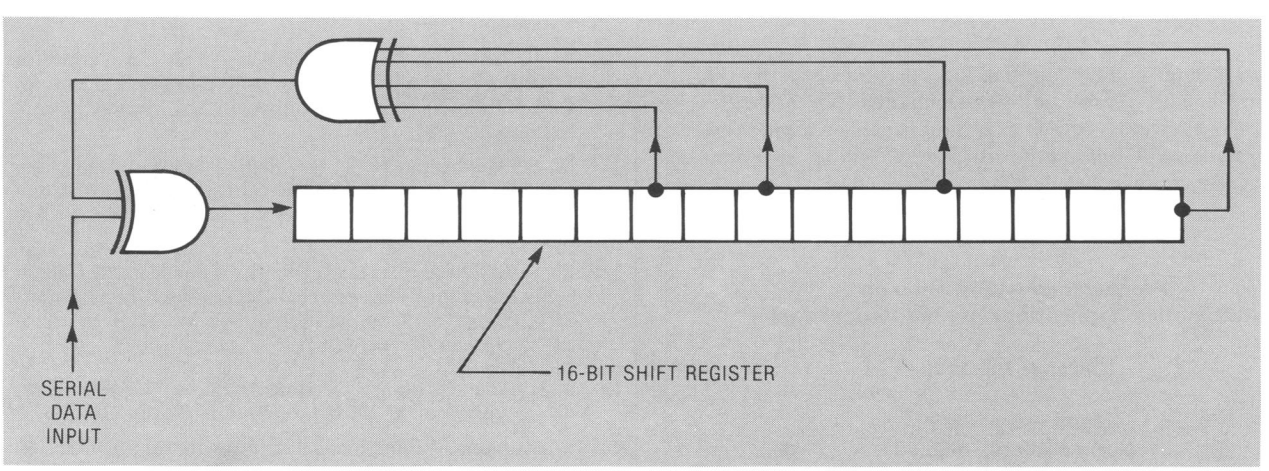
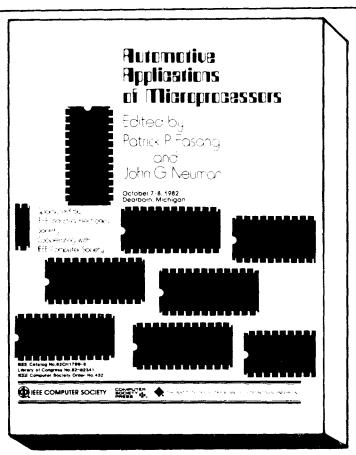


Figure 9. The 16-bit linear feedback shift register used in signature analysis.

The rapid growth of the complexity and performance of digital circuits presents a testing problem of increasing severity. Although many testing methods have worked well for SSI and MSI circuits, most of them are rapidly becoming obsolete. New techniques are required to cope with the vastly more complicated LSI circuits.

In general, testing techniques fall into the concurrent and explicit categories. In this article, we gave special attention to explicit testing techniques, especially those approaching the problem at the functional level. The explicit testing process can be partitioned into three steps: generating the test, applying the test to the UUT, and evaluating the UUT's responses. The various testing techniques are distinguished by the methods they use to perform these three steps. Each of these techniques has certain strengths and weaknesses.

We have tried to emphasize the range of testing techniques available, and to highlight some of the milestones in the evolution of LSI testing. The details of an individual test method can be found in the sources we have cited. ■



The proceedings contain tutorials on microprocessor use in electronic engine control and microcomputer applications in automotive electronics as well as session papers on engine and drive train control, automotive accessories, and test, service, and diagnostics are included. 135 pp.

Order #432

Proceedings—Workshop on Automotive Applications of Microprocessors

October 7-8, 1982

Members—$15.00
Nonmembers—$30.00
Use order form on p. 87.

References

1. M. A. Breuer and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems,* Computer Science Press, Washington, DC, 1976.
2. J. P. Hayes and E. J. McCluskey, "Testing Considerations in Microprocessor-Based Design," *Computer,* Vol. 13, No. 3, Mar. 1980, pp. 17-26.
3. J. Wakerly, *Error Detecting Codes, Self-Checking Circuits and Applications,* American Elsevier, New York, 1978.
4. D. B. Armstrong, "On Finding a Nearly Minimal Set of Fault Detection Tests for Combinatorial Nets," *IEEE Trans. Electronic Computers,* Vol. EC-15, No. 2, Feb. 1966, pp. 63-73.
5. J. P. Roth, W. G. Bouricius, and P. R. Schneider, "Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits," *IEEE Trans. Electronic Computers,* Vol. EC-16, No. 5, Oct. 1967, pp. 567-580.
6. S. B. Akers, "Test Generation Techniques," *Computer,* Vol. 13, No. 3, Mar. 1980, pp. 9-15.
7. E. I. Muehldorf and A. D. Savkar, "LSI Logic Testing—An Overview," *IEEE Trans. Computers,* Vol. C-30, No. 1, Jan. 1981, pp. 1-17.
8. O. H. Ibarra and S. K. Sahni, "Polynomially Complete Fault Detection Problems," *IEEE Trans. Computers,* Vol. C-24, No. 3, Mar. 1975, pp. 242-249.
9. T. Sridhar and J. P. Hayes, "Testing Bit-Sliced Microprocessors," *Proc. 9th Int'l Symp. Fault-Tolerant Computing,* 1979, pp. 211-218.
10. *The Am2900 Family Data Book,* Advanced Micro Devices, Inc., 1979.
11. Z. Kohavi, *Switching and Finite Automata Theory,* McGraw-Hill, New York, 1970.
12. S. M. Thatte, "Test Generation for Microprocessors," PhD thesis, University of Illinois, Urbana, 1979.
13. S. M. Thatte and J. A. Abraham, "Test Generation for Microprocessors," *IEEE Trans. Computers,* Vol. C-29, No. 6, June 1980, pp. 429-441.
14. M. A. Breuer and A. D. Friedman, "Functional Level Primitives in Test Generation," *IEEE Trans. Computers,* Vol. C-29, No. 3, Mar. 1980, pp. 223-235.
15. M. S. Abadir and H. K. Reghbati, "Test Generation for LSI: A New Approach," Tech. Report 81-7, Dept. of Computational Science, University of Saskatchewan, Saskatoon, 1981.
16. M. S. Abadir and H. K. Reghbati, "Test Generation for LSI: Basic Operations," Tech. Report 81-8, Dept. of Computational Science, University of Saskatchewan, Saskatoon, 1981.
17. M. S. Abadir and H. K. Reghbati, "Test Generation for LSI: A Case Study," Tech. Report 81-9, Dept. of Computational Science, University of Saskatchewan, Saskatoon, 1981.
18. M. S. Abadir and H. K. Reghbati, "Functional Testing of Semiconductor Random Access Memories," Tech. Report 81-6, Dept. of Computational Science, University of Saskatchewan, Saskatoon, 1981.
19. S. B. Akers, "Binary Decision Diagram," *IEEE Trans. Computers,* Vol. C-27, No. 6, June 1978, pp. 509-516.
20. S. B. Akers, "Functional Testing with Binary Decision Diagram," *Proc. 8th Int'l Symp. Fault-Tolerant Computing,* June 1978, pp. 82-92.

21. B. A. Zimmer, "Test Techniques for Circuit Boards Containing Large Memories and Microprocessors," *Proc. 1976 Semiconductor Test Symp.*, pp. 16-21.

22. P. Agrawal and V. D. Agrawal, "On Improving the Efficiency of Monte Carlo Test Generation," *Proc. 5th Int'l Symp. Fault-Tolerant Computing,* June 1975, pp. 205-209.

23. D. Bastin, E. Girard, J. C. Rault, and R. Tulloue, "Probabilistic Test Generation Methods," *Proc. 3rd Int'l Symp. Fault-Tolerant Computing,* June 1973, p. 171.

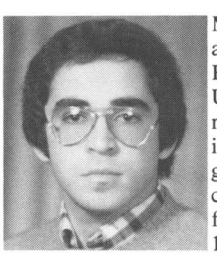
24. J. P. Hayes, "Transition Count Testing of Combinational Logic Circuits," *IEEE Trans. Computers,* Vol. C-25, No. 6, June 1976, pp. 613-620.

25. "Signature Analysis," *Hewlett-Packard J.,* Vol. 28, No. 9, May 1977.

26. R. David, "Feedback Shift Register Testing," *Proc. 8th Int'l Symp. Fault-Tolerant Computing,* June 1978.

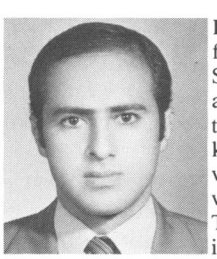
27. H. J. Nadig, "Testing a Microprocessor Product Using Signature Analysis," *Proc. 1978 Semiconductor Test Symp.*, pp. 159-169.

28. J. B. Peatman, *Digital Hardware Design,* McGraw-Hill, New York, 1980.



Magdy S. Abadir is a research assistant and graduate student working towards the PhD degree in electrical engineering at the University of Southern California. His research interests include functional testing, design for testability, test pattern generation, and design automation. He received the BSc degree in computer science from Alexandria University, Egypt, in 1978 and the MSc in computer science from the University of Saskatchewan in 1981.

Abadir's address is the Department of Electrical Engineering, University of Southern California, Los Angeles, CA 90007.



Hassan K. Reghbati is an assistant professor in the Department of Computing Science at Simon Fraser University, Burnaby, British Columbia. He was an assistant professor at the University of Saskatchewan from 1978 to 1982, where he was granted tenure. From 1970 to 1973 he was a lecturer at Arya-Mehr University of Technology, Tehran, Iran. His research interests include fault-tolerant computing, VLSI systems, design automation, and computer communication. The author or coauthor of over 15 papers, he has published in *IEEE Micro, Computer, Infor,* and *Software—Practice & Experience.* One of his papers has been reprinted in the *Auerbach Annual 1980: Best Computer Papers.* He has served as a referee for many journals and was a member of the program committee of the CIPS '82 National Conference.

Reghbati holds a BSc from Arya-Mehr University of Technology and an MSc in electrical engineering from the University of Toronto, where he is completing requirements for his PhD. He is a member of the IEEE.

His address is the Department of Computing Science, Simon Fraser University, Burnaby, BC V5A 1S6, Canada.

Harper & Row In 1983.

Mohamed Rafiquzzaman

California State Polytechnic University—Pomona

MICROPROCESSORS AND MICROCOMPUTER DEVELOPMENT SYSTEMS

Designing Microprocessor-Based Systems

Familiarizes students with the basic concepts of typical 8, 16, and 32-bit microprocessors, interface chips, and microcomputer development systems necessary to design and develop hardware and software for microprocessor-based applications. Contains numerous laboratory exercises and examples designed for practical applications.

4/83. 640 pages(tentative). Solutions Manual.

Sydney B. Newell

Microelectronics Center, North Carolina

INTRODUCTION TO MICROCOMPUTING

Provides an introduction to microprocessors and assembly language programming using the 6800 and 68000.

1982. 615 pages. Solutions Manual.

Coming soon—

A new Harper & Row series in Computer Engineering

Consulting Editor, Daniel P. Siewiorek, Carnegie-Mellon University

If you plan to write on the subjects covered by the new series, your ideas and manuscripts will be most welcome and will receive full consideration. Please contact Carl McNair, Harper & Row computer engineering editor, for additional information.

To request examination copies, write to Suite 3D, Harper & Row, 10 East 53d Street, New York, N.Y. 10022. Please include course title, enrollment, and present text.

Harper & Row

Reader Service Number 4